Exercises

V.1 Given is a collection of tasks that modify shared variable *s* through statements of the form

$$s := s+E$$

where *E* is the outcome of an expression and can be positive or negative. It is given that *s* is initially 0. The program must maintain: $s \geq 0$

a. Make a data structure containing both *s* and the variable and mutex required for exclusion, and synchronize the tasks with conditional critical sections using condition variables to maintain the synchronization invariant.

b. Make a function to update *s* so that the above statements can be replaced by a call to this function.

Exercises

V.2 Provide an implementation of a general semaphores using condition variables, i.e., implement the data structure, and the P and V operations on that data structure. Try to avoid "*Sigall".* Can you make the implementation fair?

V.3 Synchronize the code of two consumers and one producer accessing a bounded buffer of depth *M*. Assume that the first consumer consumes three items from the buffer and the second consumers always consumes four items. The producer produces *L* items each time. Use condition variables for the synchronization.

V.4 Redo exercise A.6 using condition variables. Answer the same questions. What about starvation?

Exercises

M.1 Make a monitor implementation of a semaphore

M.2 In the readers/writers problem, add additional synchronization (introduce new variables) to change the behavior as follows

(a) no reader may enter while a writer is waiting

(b) at most *K* new readers may enter while a writer is waiting

(c) a writer has preference but at most 5 may go in a row; after that, all waiting readers go first but no new readers if a writer is still waiting

TU/e

Answers to exercises 2IN05: V.1a

```
type shared_int =
 record
    v : int
    c : condition {v ¸ 0}
    m: Mutex init 1
 end

var s : shared_int
```

```
with s do
  P(m);
  while v+E < 0 do
     Wait (m, c)
  od ; {v+E ¸ 0}
  v := v+E;
  Sigall (c);
  V(m)
od
```

Implementation of CCS can be simplified by making case distinction between $E < 0$ and $E \geq 0$

TU/e

Answers to exercises 2IN05: V.1b

```
proc Update (var s: shared_int, e: int)
|[ with s do
    P(m);
    while v+e < 0 do
       Wait (m, c)
    od ; {v+e ¸ 0}
    v := v+e;
    Sigall (c);
    V(m)
  end
]|
```

Answers to exercises 2IN05: V.2

```
type Semaphore =
  record
    v : int
    c : condition { v ¸ 0 }
    m: Mutex init 1
  end

var s : Semaphore
```

```
proc VV
(var s: Semaphore)=
|[ with s do
    P(m);
    v := v+1;
    Signal (c);
    V(m)
  od
]|
```

```
proc PP
(var s: Semaphore)=
|[ with s do
    P(m);
    while v = 0 do
      Wait (m, c)
    od ;
    v := v-1;
    V(m)
  od
]|
```

Only fair when signaling is fair, i.e., no pending *PP*-operation is permanently neglected, and when *m* is a strong mutex

Answers to exercises 2IN05: V.2 (fair)

```
type Semaphore =
  record
    v : int
    c : condition { v ¸ 0 }
    m: Mutex init 1
    f, l: int init 0, 0
    { inv: 0 · f · l}
  end
{ f = c PP outside CS
  l-f = q PP outside CS }
```

procedure *VV* uses *Sigall*

```
proc PP
(var s: Semaphore)=
|[ var t: int;
  with s do
    P(m);
    t := l; l := l+1;
    while v = 0 Ç f < t do
      Wait (m, c)
    od ;
    v := v-1; f := f+1
    V(m)
  od
]|
```

Pending PP-operations are handled on a first-come-first-serve basis

variable f indicates the pending PP-operation (if any) whose turn it is.

m strong

V.3(i): Model

Assume buffersize $M \geq 1$. Assume that the producer produces L messages at time.

```
Proc ConsumerA =
|[ while true do
   { st ≥ 3 }
   A: st := st-3;
  od
]|
```

```
Proc ConsumerB =
|[ while true do
   { st ≥ 4 }
   B: st := st -4;
  od
]|
```

```
Proc Producer =
|[ while true do
   { st+L ≤ M }
   C: st := st+L;
  od
]|
```

Buffer manipulation has been left out.
Places where a condition must hold have been identified

V.3(ii): Standard solution

Apply condition synchronization principle

Beware:
Wait (m, ca) equals *<V(m);Wait(ca)>;P(m)*

```
var ca, cb, cc: Condition;
    m: Semaphore init 1;
    st: int;
```

```
A: P(m);
   while st < 3 do
   Wait (m, ca)
   od ;
   st := st-3;
   Signal (cc);
   V(m);
```

```
B: P(m);
   while st < 4 do
   Wait (m, cb)
   od ;
   st := st-4;
   Signal (cc);
   V(m);
```

```
C: P(m);
   while st+L > M do
   Wait (m, cc)
   od;
   st := st+L;
   if st ≥ 4 then Signal (cb) fi;
   if st ≥ 3 then Signal (ca) fi;
   V(m)
```

V.3(ii): properties of standard solution

- Deadlock, when $L > M$
- No deadlock, when $M \geq L+3$
- Fair, when m is a strong semaphore and it is guaranteed that $st > 3$ occurs infinitely often, e.g. because $L > 3$. Otherwise *ConsumerA* can starve *ConsumerB* even when m is strong.
- Minimal waiting: in the sense that processes are only blocked in case of buffer underflow or overflow would occur.
- When L is large, then the consumers can consume multiple times before becoming blocked again.
- Variations are possible:
  - 1 condition variable for both consumers with *Sigall* signaling

V.3(iii): Indicate waiting (interest)

```
var ca, cb, cc: Condition;
    m: Semaphore init 1;
    st: int; qA, qB: bool
```

```
A: P(m);
   while st < 3 do
    qA := true;
    Wait (m, ca);
    qA := false
   od ;
   st := st-3;
   Signal (cc);
   V(m);
```

B analogous to A

Signal only waiting consumers

```
C: P(m)
   while st+L > M do
   Wait (m, cc)
   od;
   st := st+L;
   if st ≥ 4 Æ qB then Signal (cb) fi;
   if st ≥ 3 Æ qA then Signal (ca) fi;
   V(m)
```

Fairness issues remain unaltered

V.3(iv): Fair solution using interest and turns

TU/e

```
var ca, cb, cc: Condition;
    m: Semaphore init 1;
    st: int; qA, qB: bool;
    t: enum (A, B);
```

```
A: P(m);
   while st < 3 do
    qA := true;
    Wait (m, ca);
    qA := false
   od ;
   st := st-3;
   Signal (cc); V(m);
```

```
C: P(m)
   while st+L > M do
   Wait (m, cc)
   od;  st := st+L;
   if st ≥ 4 Æ (t = B Ç : qA) then
     t := A; Signal (cb)
  elif st ¸ 4 Æ (t = A Ç : qB) then
     t := B; Signal (ca)
   fi;
   if st ≥ 3 Æ qA then
     Signal (ca)
   fi;
   V(m)
```

Answers to exercises 2IN05: V.4(i)

TU/e

Equip the philosopher procedure with
entry protocol: StartEating (n: int)
exit protocol: StopEating (n: int)

```
Proc Philosopher (n: 0 <= n < N) =
|[ while true do
   StartEating (n);   // Obtain forks
   Eat (n);           // Critical section
   StopEating (n);    // Release forks
   Think (n)          // Non-critical section
  od
]|
```

TU/e

Answers to exercises 2IN05: V.4(ii)

Introduce one condition variable per philosopher.
Allows precise signaling.
All condition variables share the same semaphore.

```
var IsEating: array [0..N) of boolean initially all false;
        f: array [0..N) of Semaphore initially all 1;
        cv: array [0..N) of Condition;
        m: Semaphore initially 1;
```

TU/e

Answers to exercises 2IN05: V.4(iii)

```
Proc StartEating (n: 0 <= n < N)
=
|[ P(m);
  while IsEating [n ª 1]
        Ç IsEating [n © 1]
  do Wait (m, cv[n])
  od;
  P(f[n]);     // grab left fork
  P(f[n © 1]); // grab right fork
  IsEating[n] := true;
  V(m);   // no signaling required
]|
```

```
Proc StopEating (n: 0 <= n < N) =
|[ P(m);
  IsEating[n] := false;
  V(f[n]);     // release left fork
  V(f[n © 1]); // release right fork
  Signal (cv[n ª 1]); // equals Sigall
  Signal (cv[n © 1]); // equals Sigall
  V(m);
]|
```

Are the f-semaphores necessary?

Answers to exercises 2IN05: V.4(iv)

- Minimal waiting:
  No philosopher is blocked because another has possession of a fork while not eating.

- Fairness:
  The solution is not fair because philosophers 0 and 2 can starve philosopher 1 by taking turns.
  Philosopher 0 blocks philosopher 1 on fork 1 and philosopher 2 blocks philosopher 1 on fork 2.
  Philosophers 0 and 2 take turns on fork 0.
  Changing the order of signaling does not help.

Answers to exercises 2IN05: M.1(i)

```
Monitor Semaphore =
|[ var v : int;
    Pblock { v > 0 } : Condition;

  proc P = |[ while v = 0 do Wait (Pblock) od; v := v-1 ]| ;

  proc V = |[ v := v+1; Sigall (Pblock) ]|;

  v := v0
]|
```

Answers to exercises 2IN05: M.2a

Modify the ReaderWriter monitor as follows:
- add **var** *nww: int* { number of waiting writers } **init** *0*
- replace procedures *Rentry* and *Wentry*

```
proc Rentry =
[ while nw > 0 Ç nww > 0
    do
    Wait (Rblock)
  od ;{ nw = 0 Æ nww = 0 }
  nr := nr+1;
  MonExit
]
```

```
proc Wentry =
[ while nw > 0 Ç nr > 0 do
    nww := nww+1;
    Wait (Wblock);
    nww := nww-1
  od ;{ nw = 0 Æ nr = 0 }
  nw := nw+1;
  MonExit
]
```


Answers to exercises 2IN05: M.2b(i)

Modify the ReaderWriter monitor of M.2a as follows:
- add **var** *nreww: int* { nr of readers entered while writers are waiting} **init** *0*
- replace procedures *Rentry* and *Wentry*

```
proc Rentry =
[ while nw > 0 Ç  (nww > 0 Æ nreww ¸ 3) do
    Wait (Rblock)
  od ;{ nw = 0 Æ (nww = 0  Ç nreww < 3) }
  nr := nr+1;
  if nww > 0 then nreww := nreww+1 fi ;
  MonExit
]
```

TU/e

Answers to exercises 2IN05: M.2b(ii)

```
proc Wentry =
[ while nw > 0 Ç nr > 0 do
    nww := nww+1;
    Wait (Wblock);
    nww := nww-1;
    if nww = 0 then nreww := 0 fi
  od ;{ nw = 0 Æ nr = 0 }
  nw := nw+1;
  MonExit
]
```

TU/e

Answers to exercises 2IN05: M.2c(i)

Modify the ReaderWriter monitor of M.2a as follows:

- add **var** *ncw: int* { nr of consecutive writers } **init** *0*
- add **var** *nwr5: int* { number of waiting readers at *ncw* = *5* } **init** *0*
- add **var** *icRentry: int* { number of initiated calls to Rentry } **init** *0*
- add **var** *ccRentry: int* { number of completed calls to Rentry } **init** *0*
  - overflow issues regarding *icRentry* and *ccRentry* ignored
- replace procedures *Rentry* and *Wentry*

All reader get a number *cn* in the Rentry protocol . In case there have been *5* consecutive writers, the number of reader that have entered in the mean time are given by

$ccRentry \cdot cn < ccRentry + icRentry$

TU/e

Answers to exercises 2IN05: M.2c(ii)

```
proc Rentry =
[[ var cn: int {call number}
  cn := icRentry; icRentry := icRentry+1;
  while nw > 0 Ç
        (nww > 0 Æ (ncw < 5 Ç (cn ¸ ccRentry+nwr5 Æ nwr5 > 0))) do
    Wait (Rblock)
  od ;{ nw = 0 Æ
        (nww = 0 Ç (ncw ¸ 5 Æ (cn < ccRentry+nwr5 Ç nwr5 = 0)) ) }
  nr := nr+1;
  ccRentry := ccRentry +1;
  if ncw ¸ 5 Æ nwr5 > 0 then nwr5:= nwr5-1 fi;
  if nwr5 = 0 then ncw := 0 fi ;
  MonExit
]]
```

SAN

TU/e

Answers to exercises 2IN05: M.2c(iii)

```
proc Wentry =
[[ while nw > 0 Ç nr > 0 Ç ncw ¸ 5 do
     nww := nww+1;
     Wait (Wblock);
     nww := nww-1;
   od ;{ nw = 0 Æ nr = 0 Æ ncw < 5}
   nw := nw+1; ncw := ncw+1;
   if ncw = 5 then nwr5 := icRentry - ccRentry fi ;
   MonExit
]]
```

Since only readers can reset *ncw* to *0* this solution will never allow more than 5 consecutive writers, even when a 6-th writer in a row arrives at a moment in time when no readers are interested to enter

SAN

Question: Does this solutions work for all signaling strategies?